深入浅出Flink(4)

一、课前准备

掌握上节课内容

二、课堂主题

深入理解Checkpoints机制原理，了解Flink的容错思路

三、课程目标

2. 掌握Checkpoints算法
3. 掌握Checkpoints配置使用
4. 了解Flink的容错

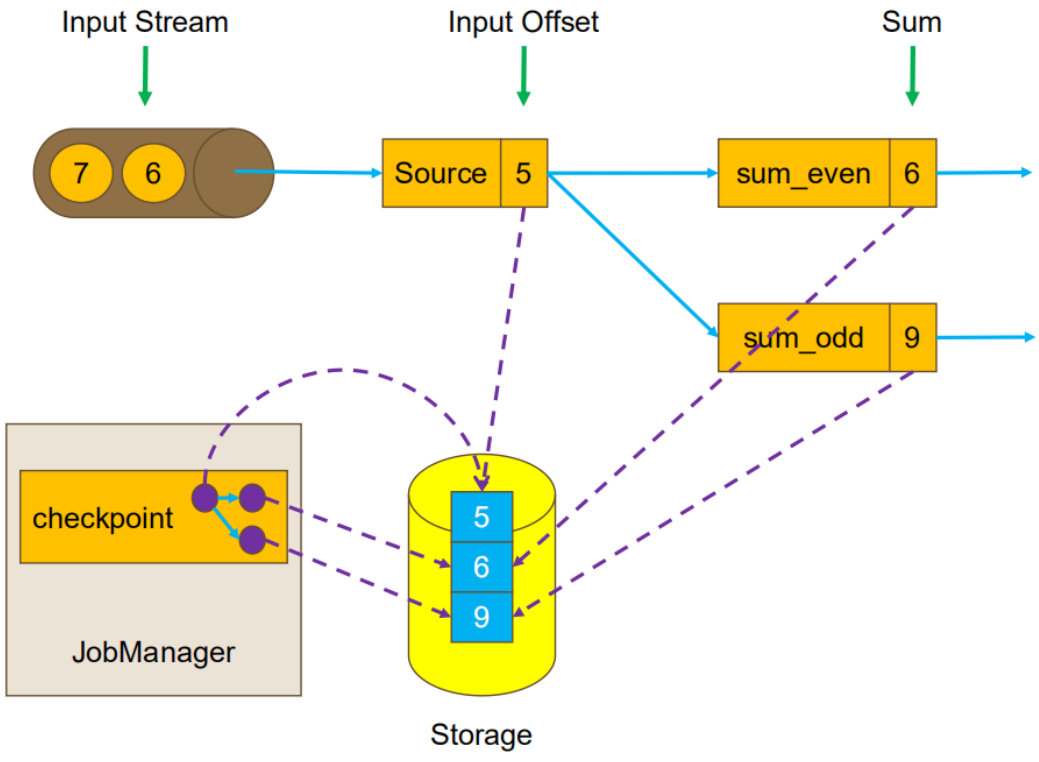
四、知识要点

4.1 Checkpoint原理

4.1.1 Checkpoint概述

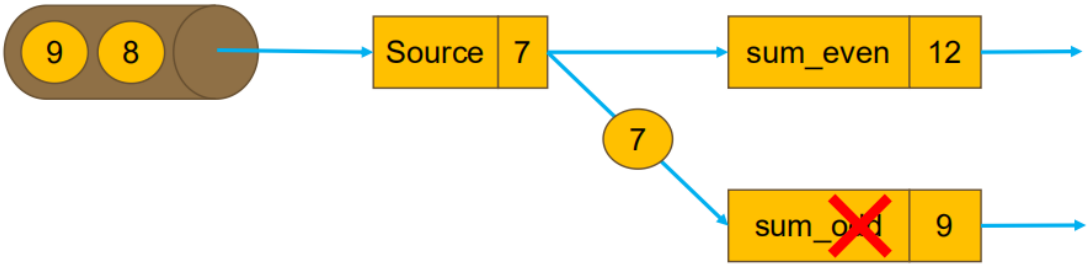
checkpoint机制是Flink可靠性的基石，可以保证Flink集群在某个算子因为某些原因(如 异常退出)出现故障时，能够将整个应用流图的状态恢复到故障之前的某一状态，保 证应用流图状态的一致性。

4.1.2 Checkpoint的简单想法

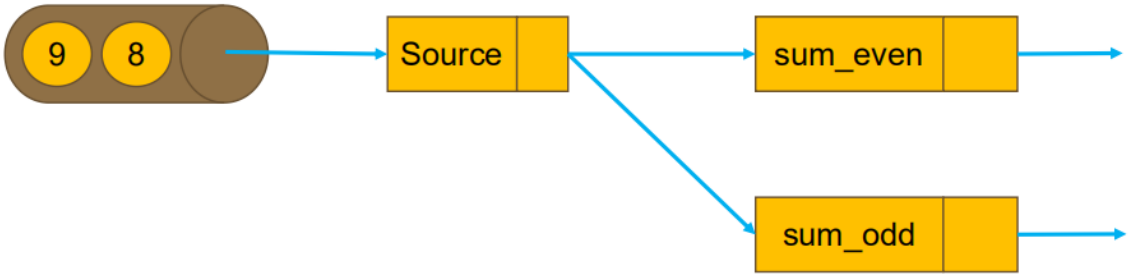


4.1.3 Checkpoint 恢复流程

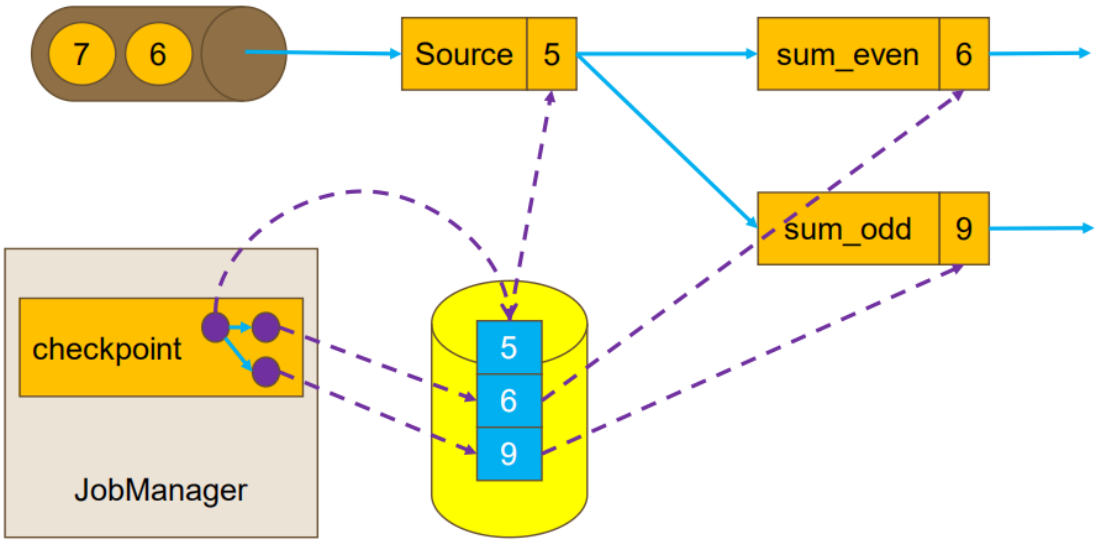
任务运行失败



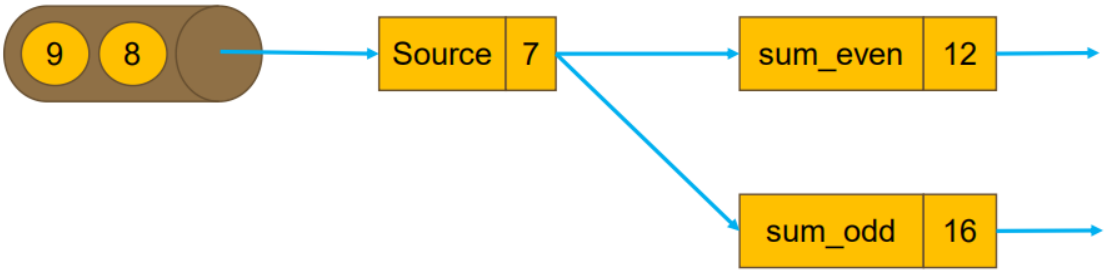
重启应用



从checkpoint恢复数据

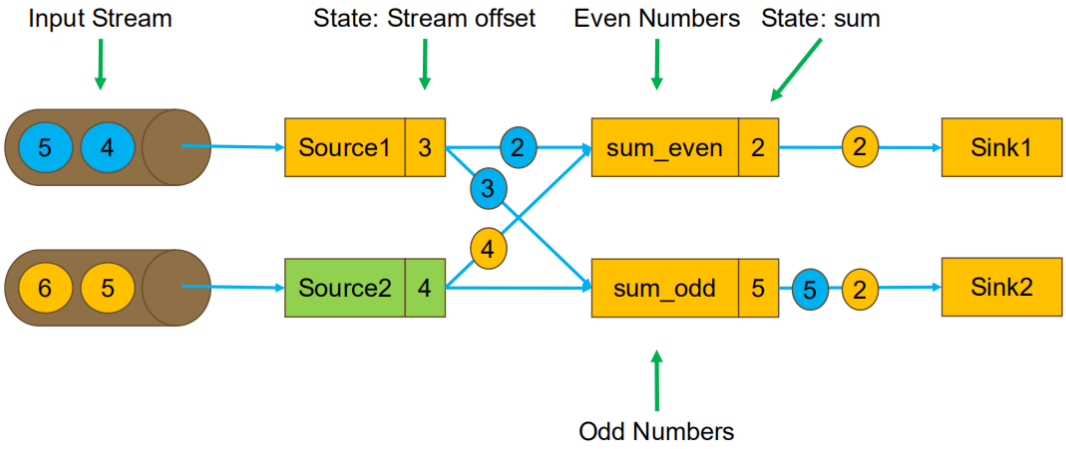


任务继续运行



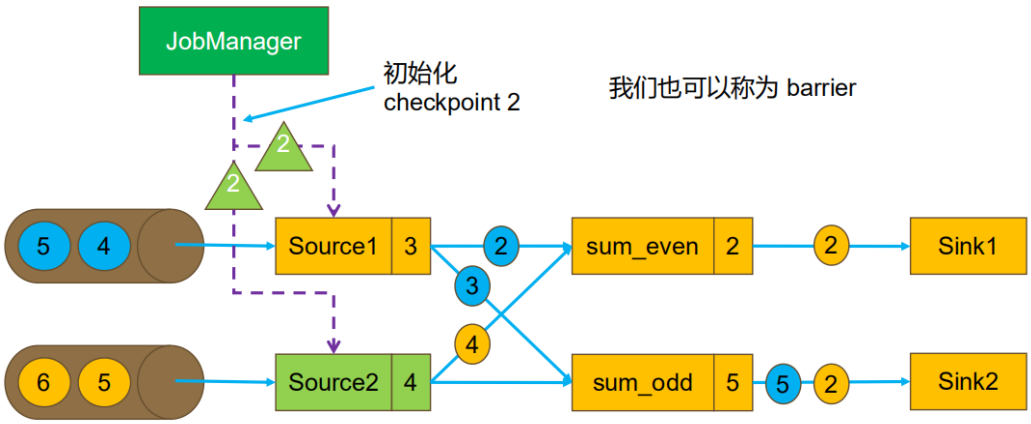
4.1.3 Chandy-Lamport 算法

1. 任务开启

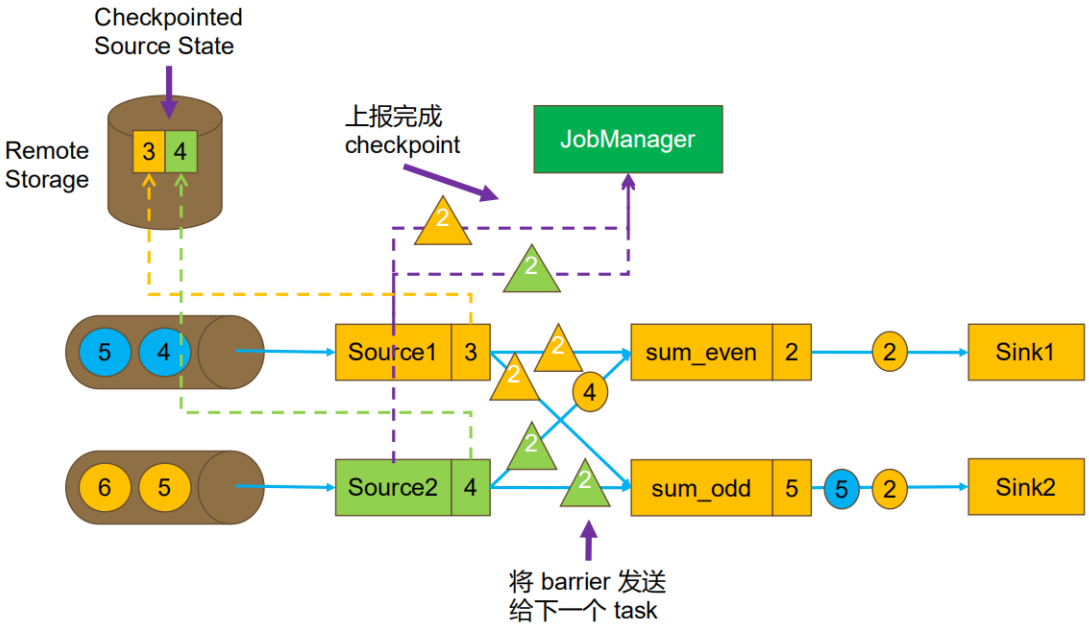


2. JobMananger发起Checkpoint

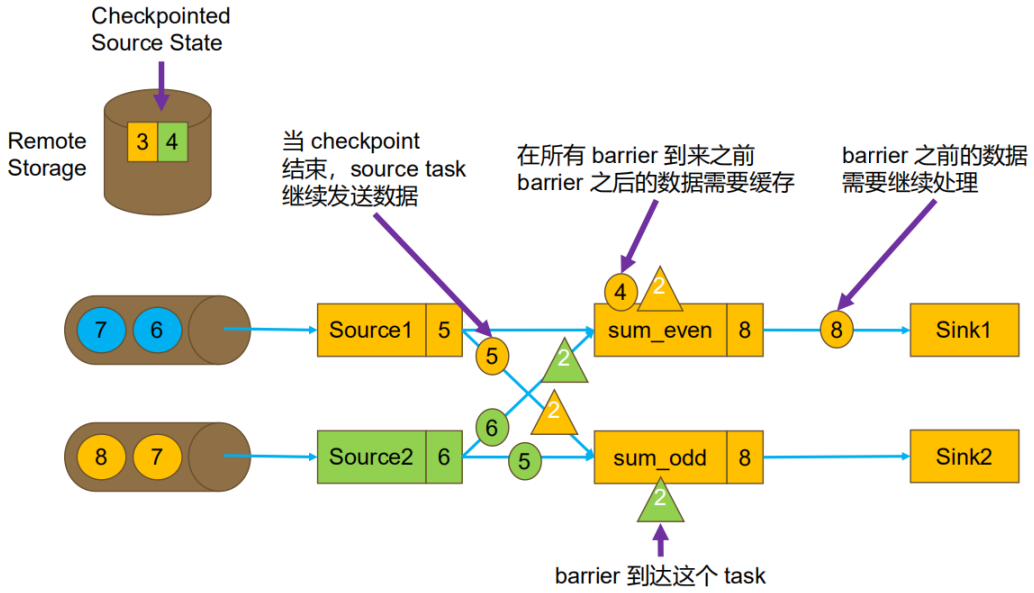
3.



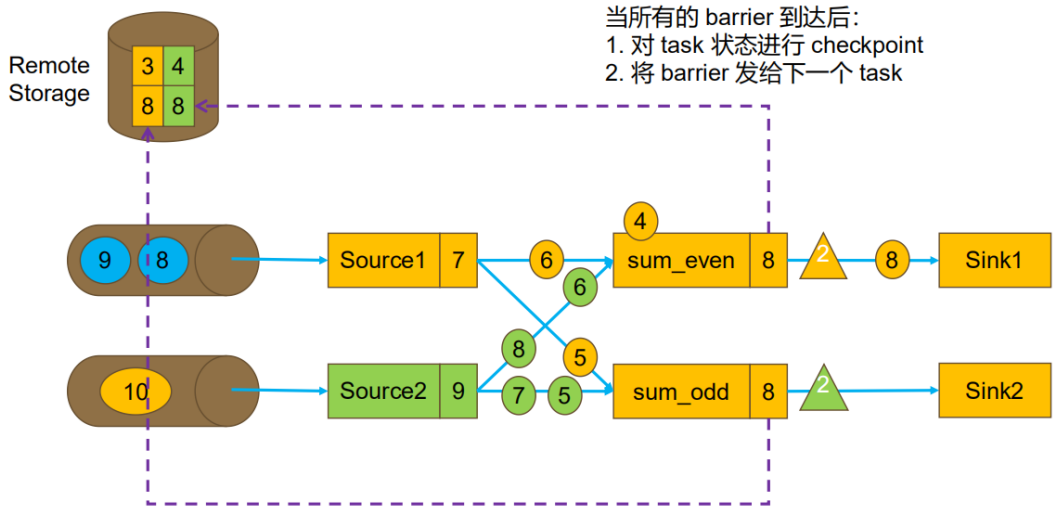
3.source上报checkpoint



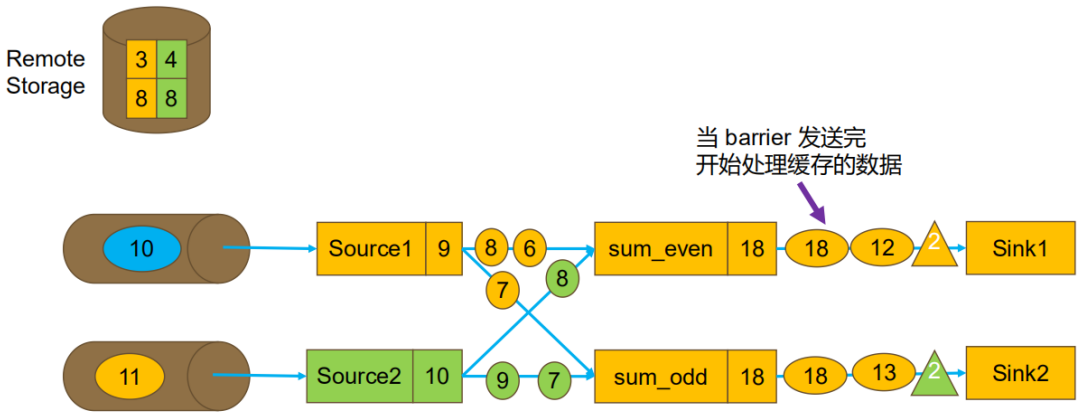
4. 数据处理



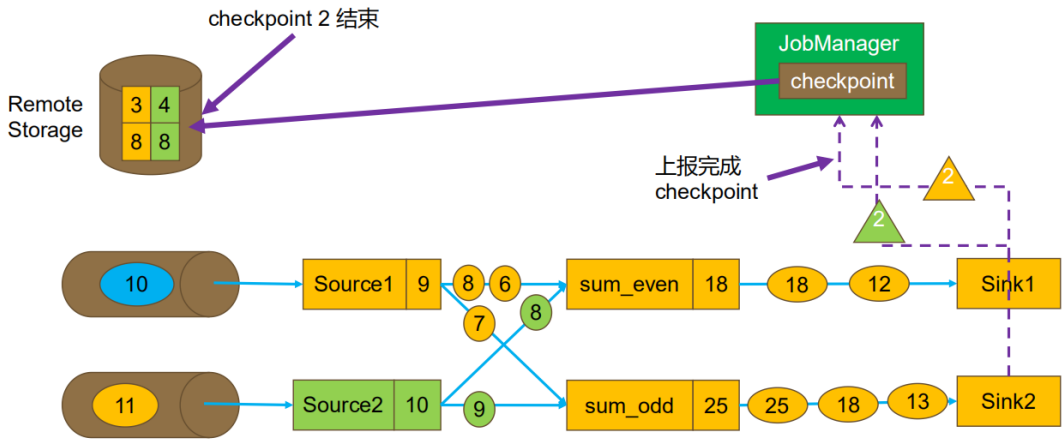
5. barrier对齐



6. 缓存数据处理



7. sink上报checkpoint



4.2 Checkpoint配置

默认checkpoint功能是disabled的，想要使用的时候需要先启用，checkpoint开启之后，checkPointMode有两种，Exactly-once和At-least-once，默认的checkPointMode是Exactly-once，Exactly-once对于大多数应用来说是最合适的。At-least-once可能用在某些延迟超低的应用程序（始终延迟为几毫秒）。

```
默认checkpoint功能是disabled的，想要使用的时候需要先启用
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
// 每隔1000 ms进行启动一个检查点【设置checkpoint的周期】
env.enableCheckpointing(1000);
// 高级选项：
// 设置模式为exactly-once （这是默认值）
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
// 检查点必须在一分钟内完成，或者被丢弃【checkpoint的超时时间】
env.getCheckpointConfig().setCheckpointTimeout(60000);
// 同时允许多少个checkpoint
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
// 确保检查点之间有至少500 ms的间隔【checkpoint最小间隔】
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);
// 表示一旦Flink处理程序被cancel后，会保留Checkpoint数据，以便根据实际需要恢复到指定的
Checkpoint【详细解释见备注】
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckpointCl
eanup.RETAIN_ON_CANCELLATION);
```

4.3 重启策略

4.3.1 重启策略概述

Flink支持不同的重启策略，以在故障发生时控制作业如何重启，集群在启动时会伴随一个默认的重启策略，在没有定义具体重启策略时会使用该默认策略。 如果在工作提交时指定了一个重启策略，该策略会覆盖集群的默认策略，默认的重启策略可以通过 Flink 的配置文件 flink-conf.yaml 指定。配置参数 restart-strategy 定义了哪个策略被使用。

常用的重启策略

（1）固定间隔 (Fixed delay)

（2）失败率 (Failure rate)

（3）无重启 (No restart)

如果没有启用 checkpointing，则使用无重启 (no restart) 策略。

如果启用了 checkpointing，但没有配置重启策略，则使用固定间隔 (fixed-delay) 策略， 尝试重启次数

默认值是：Integer.MAX_VALUE，重启策略可以在flink-conf.yaml中配置，表示全局的配置。也可以在应用代码中动态指定，会覆盖全局配置。

4.3.2 重启策略

固定间隔 (Fixed delay)

```
第一种：全局配置 flink-conf.yaml
restart-strategy: fixed-delay
restart-strategy.fixed-delay.attempts: 3
restart-strategy.fixed-delay.delay: 10 s
第二种：应用代码设置
env.setRestartStrategy(RestartStrategies.fixedDelayRestart(
  3, // 尝试重启的次数
  Time.of(10, TimeUnit.SECONDS) // 间隔
));
```

失败率 (Failure rate)

```
第一种：全局配置 flink-conf.yaml
restart-strategy: failure-rate
restart-strategy.failure-rate.max-failures-per-interval: 3
restart-strategy.failure-rate.failure-rate-interval: 5 min
restart-strategy.failure-rate.delay: 10 s
第二种：应用代码设置
env.setRestartStrategy(RestartStrategies.failureRateRestart(
  3, // 一个时间段内的最大失败次数
  Time.of(5, TimeUnit.MINUTES), // 衡量失败次数的是时间段
  Time.of(10, TimeUnit.SECONDS) // 间隔
));
```

无重启 (No restart)

```
第一种：全局配置 flink-conf.yaml
restart-strategy: none
第二种：应用代码设置
env.setRestartStrategy(RestartStrategies.noRestart());
```

4.3.3 多checkpoint

默认情况下，如果设置了Checkpoint选项，则Flink只保留最近成功生成的1个Checkpoint，而当Flink程序失败时，可以从最近的这个Checkpoint来进行恢复。但是，如果我们希望保留多个Checkpoint，并能够根据实际需要选择其中一个进行恢复，这样会更加灵活，比如，我们发现最近4个小时数据记录处理有问题，希望将整个状态还原到4小时之前Flink可以支持保留多个Checkpoint，需要在Flink的配置文件conf/flink-conf.yaml中，添加如下配置，指定最多需要保存Checkpoint的个数：

```
state.checkpoints.num-retained: 20
```

这样设置以后就查看对应的Checkpoint在HDFS上存储的文件目录
hdfs dfs -ls hdfs://namenode:9000/flink/checkpoints
如果希望回退到某个Checkpoint点，只需要指定对应的某个Checkpoint路径即可实现

4.3.4 从checkpoint恢复数据

如果Flink程序异常失败，或者最近一段时间内数据处理错误，我们可以将程序从某一个Checkpoint点进行恢复

```
flink run -s
hdfs://192.168.123.102:9000/flink/checkpoint/40fc6b85c66a6c36a1c9f66cf21f14fb/chk-14/_metadata  -c com.nx.stream.state.lesson04.TestCheckpoint nx-flink-1.0-SNAPSHOT.jar --hostname 192.168.123.102 --port 9999
```

程序正常运行后，还会按照Checkpoint配置进行运行，继续生成Checkpoint数据。

当然恢复数据的方式还可以在自己的代码里面指定checkpoint目录，这样下一次启动的时候即使代码发生了改变就自动恢复数据了。

4.4 SavePoint

SavePoint是一个重量级的Checkpoint,你可以把它当做在某个时间点程序状态全局镜像，以后程序在进行升级，或者修改并发度等情况，还能从保存的状态位继续启动恢复。可以保存数据源offset，operator操作状态等信息，可以从应用在过去任意做了savepoint的时刻开始继续消费。

```
Checkpint和SavePoint

概念：Checkpoint 是 自动容错机制 ，Savepoint 程序全局状态镜像 。
目的： Checkpoint 是程序自动容错，快速恢复 。Savepoint是 程序修改后继续从状态恢复，程序升级等。
用户交互:Checkpoint 是 Flink 系统行为 。Savepoint是用户触发。
状态文件保留策略：Checkpoint默认程序删除，可以设置CheckpointConfig中的参数进行保留 。
Savepoint会一直保存，除非用户删除 。
```

SavePoint
用户手动执行，是指向Checkpoint的指针，不会过期，在集群升级/代码迁移等情况下使用。
注意：为了能够在作业的不同版本之间以及 Flink 的不同版本之间顺利升级，强烈推荐程序员通过 uid(String) 方法手动的给算子赋予 ID，这些 ID 将用于确定每一个算子的状态范围。如果不手动给各算子指定 ID，则会由 Flink 自动给每个算子生成一个 ID。只要这些 ID 没有改变就能从保存点（savepoint）将程序恢复回来。而这些自动生成的 ID 依赖于程序的结构，并且对代码的更改是很敏感的。因此，强烈建议用户手动的设置 ID。

savepoint的使用

```
1：在flink-conf.yaml中配置Savepoint存储位置
不是必须设置，但是设置后，后面创建指定Job的Savepoint时，可以不用在手动执行命令时指定Savepoint的位置
state.savepoints.dir: hdfs://namenode:9000/flink/savepoints
2：触发一个savepoint【直接触发或者在cancel的时候触发】

停止程序：bin/flink cancel -s [targetDirectory] jobId [-yid yarnAppId]【针对on yarn模式需要指定-yid参数】


3：从指定的savepoint启动job
bin/flink run -s savepointPath [runArgs]
```

4.5 Flink容错机制

source

Source	Guarantees	Notes
Apache Kafka	exactly once	Use the appropriate Kafka connector for your version
AWS Kinesis Streams	exactly once	
RabbitMQ	at most once (v 0.10) / exactly once (v 1.0)	
Twitter Streaming API	at most once	
Google PubSub	at least once	
Collections	exactly once	
Files	exactly once	
Sockets	at most once	

kafka Consumer偏移量管理

```
String topic="nxtest";
        Properties consumerProperties = new Properties();

 consumerProperties.setProperty("bootstrap.servers","192.168.152.102:9092");
        consumerProperties.setProperty("group.id","testConsumer");
        consumerProperties.setProperty("auto.offset.reset"," earliest");

        FlinkKafkaConsumer011<String> myConsumer =
                new FlinkKafkaConsumer011<>(topic, new SimpleStringSchema(),
consumerProperties);
        //是否把偏移量提交到kafka的特殊topic里，默认是true
        myConsumer.setCommitOffsetsOnCheckpoints(true);

        DataStreamSource<String> data =
env.addSource(myConsumer).setParallelism(3);
```

结论

1. 我们使用FlinkKafkaConumser,并且启用Checkpoint，偏移量会通过checkpoint保存到state里面，并且默认会写入到kafka的特殊主体中，也就是__consumer_offset

2. setCommitOffsetsOnCheckpoints 默认会true，就是把偏移量写入特殊主题中

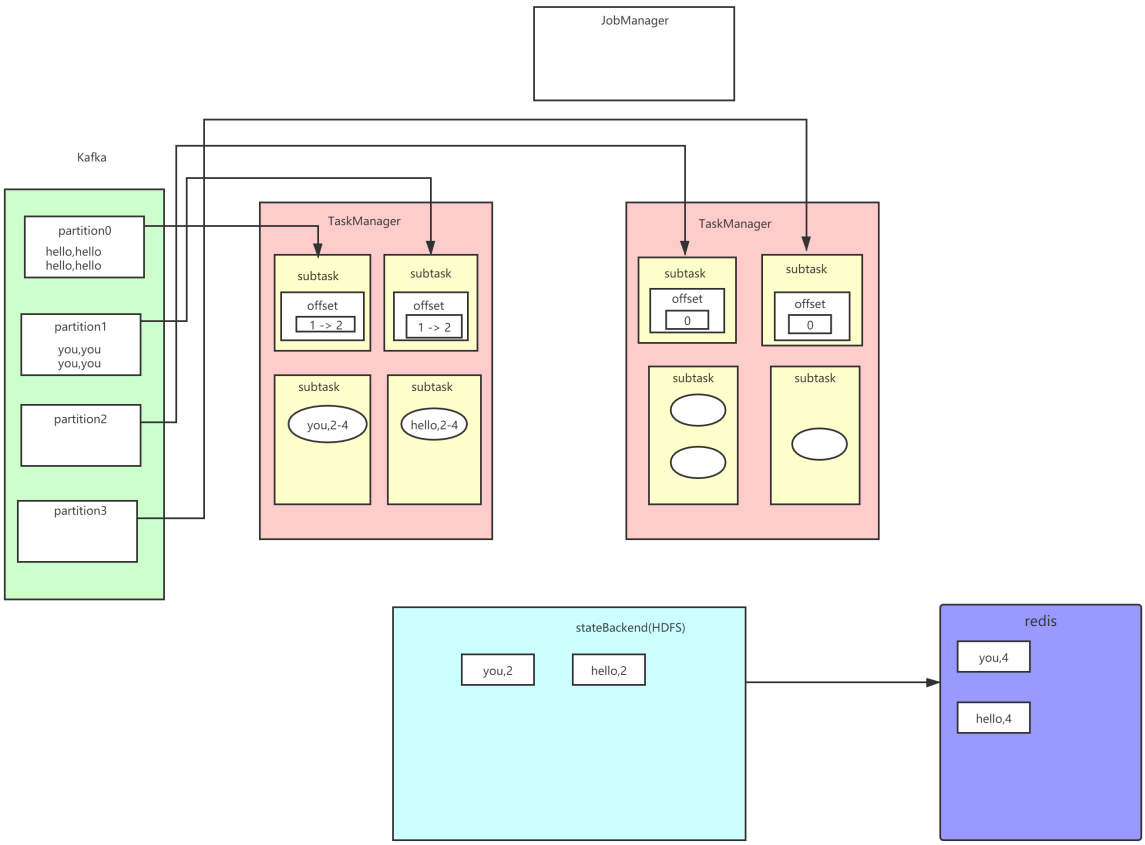
3. Flink自动重启的过程中，读取的偏移量是state中的偏移量

4. 我们手动重启任务的过程中，默认读取的是__consumer_offset中的偏移量，如果__consumer_offset里面没有那么默认就会从earliest读取数据

sink

Sink	Guarantees	Notes
HDFS BucketingSink	exactly once	Implementation depends on Hadoop version
Elasticsearch	at least once	
Kafka producer	at least once / exactly once	exactly once with transactional producers (v 0.11+)
Cassandra sink	at least once / exactly once	exactly once only for idempotent updates
AWS Kinesis Streams	at least once	
File sinks	exactly once	
Socket sinks	at least once	
Standard output	at least once	
Redis sink	at least once	

写Redis/HBase保证Exactly once方案设计



结论

> 通过幂等性实现仅一次语义

写kafka保证Exactly once

两阶段提交

在分布式系统中，可以使用两阶段提交来实现事务性从而保证数据的一致性，两阶段提交分为：预提交阶段与提交阶段，通常包含两个角色：协调者与执行者，协调者用于用于管理所有执行者的操作，执行者用于执行具体的提交操作，具体的操作流程：

1. 首先协调者会送预提交(pre-commit)命令有的执行者
2. 执行者执行预提交操作然后发送一条反馈(ack)消息给协调者
3. 待协调者收到所有执行者的成功反馈，则发送一条提交信息(commit)给执行者
4. 执行者执行提交操作

如果在流程2中部分预提交失败，那么协调者就会收到一条失败的反馈，则会发送一条rollback消息给所有执行者，执行回滚操作，保证数据一致性；但是如果在流程4中，出现部分提交成功部分提交失败，那么就会造成数据的不一致，因此后面也提出了3PC或者通过其他补偿机制来保证数据最终一致性，接下看看flink 是如何做到2PC，保证数据的一致性。

flink的流程大概与这个流程相似

flink的两阶段提交

flink中两阶段提交是为了保证端到端的Exactly Once，主要依托checkpoint机制来实现，先看一下checkpoint的整体流程，

1.JobManager会周期性的发送执行checkpoint命令(start checkpoint)；
2.当source端收到执行指令后会产生一条barrier消息插入到input消息队列中，当处理到barrier时会执行本地checkpoint，并且会将barrier发送到下一个节点，当checkpoint完成之后会发送一条ack信息给JobManager；
3. 当所有节点都完成checkpoint之后，JobManager会收到来自所有节点的ack信息，那么就表示一次完整的checkpoint的完成；
4. JobManager会给所有节点发送一条callback信息，表示通知checkpoint完成消息。接下来就可以提交事务了

对比flink整个checkpoint机制调用流程可以发现与2PC非常相似，JobManager相当于协调者，flink提供了CheckpointedFunction与CheckpointListener这样两个接口，CheckpointedFunction中有snapshotState方法，每次checkpoint触发执行方法，通常会将缓存数据放入状态中，可以理解为是一个hook，这个方法里面可以实现预提交，CheckpointListener中有notifyCheckpointComplete方法，checkpoint完成之后的通知方法，这里可以做一些额外的操作，比如真正提交kafka的事务；在2PC中提到如果对应流程2预提交失败，那么本次checkpoint就被取消不会执行，不会影响数据一致性.如果流程4失败，那么重启的时候会再次执行commit

五、 扩展

如果提交报如下的错：

```
Caused by: org.apache.flink.core.fs.UnsupportedFileSystemSchemeException: Could
not find a file system implementation for scheme 'hdfs'. The scheme is not
directly supported by Flink and no Hadoop file system to support this scheme
could be loaded.
        at
org.apache.flink.core.fs.FileSystem.getUnguardedFileSystem(FileSystem.java:450)
        at org.apache.flink.core.fs.FileSystem.get(FileSystem.java:362)
        at org.apache.flink.core.fs.Path.getFileSystem(Path.java:298)
        at org.apache.flink.runtime.state.filesystem.FsCheckpointStorage.<init>
(FsCheckpointStorage.java:64)
        at
org.apache.flink.runtime.state.filesystem.FsStateBackend.createCheckpointStorage
(FsStateBackend.java:490)
```

```
        at org.apache.flink.runtime.checkpoint.CheckpointCoordinator.<init>
(CheckpointCoordinator.java:279)
        ... 23 more
Caused by: org.apache.flink.core.fs.UnsupportedFileSystemSchemeException: Hadoop
is not in the classpath/dependencies.
        at
org.apache.flink.core.fs.UnsupportedSchemeFactory.create(UnsupportedSchemeFactor
y.java:58)
        at
org.apache.flink.core.fs.FileSystem.getUnguardedFileSystem(FileSystem.java:446)
        ... 28 more

End of exception on server side>]
        at
org.apache.flink.runtime.rest.RestClient.parseResponse(RestClient.java:390)
        at
org.apache.flink.runtime.rest.RestClient.lambda$submitRequest$3(RestClient.java:
374)
        at
java.util.concurrent.CompletableFuture.uniCompose(CompletableFuture.java:952)
        at
java.util.concurrent.CompletableFuture$UniCompose.tryFire(CompletableFuture.java
:926)
```

那么下载如下的包

下载地址：https://repo.maven.apache.org/maven2/org/apache/flink/flink-shaded-hadoop-2-uber/
要按照HADOOP版本下载，然后把jar加入flink的lib目录下，重启集群。

六、总结（5分钟）

1. 掌握Checkpoints的原理
2. 掌握Checkpoints的配置
3. 了解Flink的容错

七、作业

1. 掌握课上案例，为后面实践做准备

八、互动